J Heuristics manuscript No.
(will be inserted by the editor)

A fast greedy heuristic for scheduling modular projects

Martijn Huysmans · Kris Coolen · Fabrice Talla Nobibon · Roel Leus

Received: date / Accepted: date

Abstract This article describes a heuristic for scheduling so-called 'modular' projects. Exact solutions to this NP-hard problem can be obtained with existing branch-and-bound and dynamic-programming algorithms, but only for small to medium-size instances. The proposed heuristic, by contrast, can be used even for large instances, or when instances are particularly difficult because of their characteristics, such as a low network density. The proposed heuristic draws from existing results in the literature on sequential testing, which will be briefly reviewed. The performance of the heuristic is assessed over a dataset of 360 instances.

Keywords Scheduling · Greedy heuristics · Modular projects

1 Introduction

During research and development (R&D) projects, activities can fail to meet their expected outcomes. An activity's failure can result in the termination of the entire project, for instance when a proposed new drug fails a toxicity test. In many other cases, however, the failure of one activity is not fatal to the project, because alternative activities are available that pursue the same target. When designing a new car, for example, several slightly different designs may be 'in the running.' If the preferred design fails the wind-tunnel test, the project could simply move on to the next design. Activities that have the same target are said to constitute a *module* (following Baldwin and Clark (2000)). When one of the activities in a module is successful, the module is completed and the project can continue with the next module. In order to

Martijn Huysmans · Kris Coolen · Fabrice Talla Nobibon · Roel Leus (✉)
ORSTAT, Faculty of Business and Economics, KU Leuven, Belgium
Tel.: +32 16 32 69 67
E-mail: roel.leus@kuleuven.be

Fabrice Talla Nobibon
Strategic Planning & Engineering, FedEx Express Europe, Middle East, Indian Subcontinent & Africa

successfully complete the project, all modules need to achieve success. Only when all of the scheduled activities in a module fail, does the project fail.

In our problem statement, we will allow for precedence constraints to be specified both within and between modules. Precedence constraints between modules can be regulatory or technical in nature. Regulatory constraints often occur to protect testers or consumers; when testing a new drug, for instance, the toxicity has to be tested (e.g., via animal testing) before clinical tests on humans are allowed (De Reyck and Leus 2008). An example of a technical constraint is the impossibility of evaluating a new car design in the wind tunnel before a prototype is manufactured for this test. Precedence constraints within modules are used to model situations where one activity cannot be tried before the other. In pharmaceuticals, for example, when a module aims to prove the effectiveness of a drug, a first job may be measuring the effects after one week. If no desired effects have been found after one week, one could either decide to give up the project, or to go on to an alternative activity, for instance waiting for the results to show up after two weeks. Alternatively, trials may be repeated in different doses and with different test subjects.

Projects such as those described above will be referred to as *modular projects*. Exact scheduling algorithms for this type of projects have recently been developed by Coolen et al. (2014). Ranjbar and Davari (2013) propose exact algorithms for a special case of this problem with only one module, but the cash flows are discounted. Creemers et al. (2013) study this problem for stochastic activity durations and with discounting. Mathematical-programming models are studied by Jain and Grossmann (1999) and Schmidt and Grossmann (1996) to approximate the problem in the setting where each module contains only one job. When only few precedence constraints are imposed, however, these algorithms either run out of memory or become very slow when the number of activities increases. The main contribution of this paper is the development of a heuristic that produces 'good' schedules for such projects, i.e., schedules with a high expected profit, while requiring only very limited CPU time and computer memory. We compare the results to the optimal solutions obtained by the algorithms described in Coolen et al. (2014).

The remainder of this text is organized as follows. In Sect. 2, the modular project scheduling problem on one machine is defined. In the development of a fast heuristic for this problem, we will draw from the literature on sequential testing; we therefore include a discussion of the similarities with this branch of literature and of its most relevant references in Sect. 3. The main contribution of this text is the greedy algorithm presented in Sect. 4, and we substantiate its computational performance by means of a series of experiments that are summarized in Sect. 5. We conclude in Sect. 6.

2 Problem description

Most of the notation and definitions in this section are in line with Coolen et al. (2014).

2.1 Definitions

A *project* consists of a set of jobs $N = \{1,\ldots,n\}$ to be scheduled (the terms 'job' and 'activity' are used interchangeably), and we additionally define jobs 0 and $n+1$ as a dummy start and end job. The set of all jobs, including dummy jobs, is denoted by $\bar{N} = N \cup \{0, n+1\}$. Each job belongs to a unique module $i \in \bar{M} = \{0,1,\ldots,m+1\}$, with $\bar{M}$ the set of modules. The symbol N_i denotes the set of jobs that belong to module i. The artificial start and end jobs 0 and $n+1$ are the sole jobs of modules 0 and $m+1$, respectively: $N_0 = \{0\}$ and $N_{m+1} = \{n+1\}$. The set of non-dummy modules is denoted by $M = \{1,\ldots,m\}$. Since each job belongs to exactly one module, the set of all N_i constitutes a partition of $\bar{N}$.

Every job $k \in N$ has a probability of technical success p_k. The probability of failure will be denoted by $q_k = 1 - p_k$. The probabilities p_k are gathered in vector $\mathbf{p}$ (with p_k as its k-th component). Each job $k \in N$ also has an associated cost c_k, which is the k-th component of vector $\mathbf{c}$. The dummy end job $n+1$ has a positive *payoff* V instead of a cost; this payoff is only collected when the overall project succeeds, i.e., when for each module at least one job was successful.

The precedence constraints between the modules are defined by a strict partial order[1] $\bar{A}$ over $\bar{M}$. This strict partial order relation is a collection of predecessor-successor pairs (i,j): module j has as a prerequisite the successful execution of module i if and only if $(i,j) \in \bar{A}$. Relation $\bar{A}$ induces the relation A on M. The precedence constraints between jobs belonging to the same module i are described by the strict partial order B_i: job l in module i has as a precondition that job k in module i must be attempted first if and only if $(k,l) \in B_i$. Finally, for practical reasons, all precedence-related activity pairs are collected in one 'induced' order relation $B^* \subset N \times N$:

$$(k,l) \in B^* \quad \Leftrightarrow \quad \Big(\big(\exists i \in M : (k,l) \in B_i\big) \quad \vee \quad \big(\exists (i,j) \in A : (k \in N_i) \wedge (l \in N_j)\big) \Big).$$

With these definitions, (N, B^*) is a (strict) partially ordered set (we use the more concise term 'poset' in the remainder of this text), and the same holds for (M,A) and (N_i,B_i) for all $i \in M$. An illustration of the precedence constraints that can apply to a scheduling instance is provided in Fig. 1; we graphically represent these constraints by means of a two-layered network, where modules are rounded boxes and activities are round nodes. For this instance, $n = 5$, $m = 3$, $A = \{(1,3),(2,3)\}$, $B_1 = \{(1,2)\}$ and $B_2 = B_3 = \emptyset$. Including modules 0 and $m+1$ in the analysis, we have $\bar{A} = \{(0,1),(0,2),(0,3),(0,4),(1,3),(2,3),(1,4),(2,4),(3,4)\}$. Note that transitive arcs between modules, such as $(0,3)$, are omitted from the figure.

Essential to the problem description is the uncertainty in the activity outcomes. A *realization* or *state vector* $\mathbf{x} = (x_1,\ldots,x_n) \in \{0,1\}^n \equiv \mathbb{B}^n$ is a vector indicating for each job k whether it succeeds ($x_k = 1$) or fails ($x_k = 0$). Each component x_k is the outcome of a Bernoulli trial X_k with probability of success p_k. The random variables $X_1,\ldots,X_n$ are assumed to be mutually independent. The actual realization x_k of each X_k is known only after job k is executed. The symbol $\mathbf{X}$ represents the vector with components X_k.

[1] A strict (partial) order $O \subset V \times V$ over a set V is a relation defined on V that is both asymmetric ($(i,j) \in O$ implies $(j,i) \notin O$) and transitive ($(i,j) \in O$ and $(j,l) \in O$ implies $(i,l) \in O$).

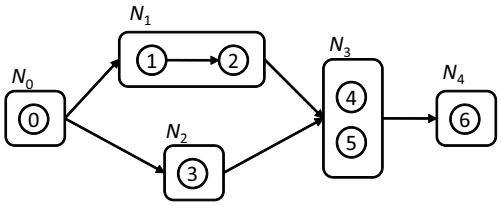


Fig. 1 Graphical representation of a modular project with seven jobs and five modules

For scheduling, the only relevant characteristics of the jobs are the following: their probability of success, their cost, and their predecessors. In this paper, we only consider 'sequential' schedules, in which jobs are processed one-at-a-time, and we therefore do not explicitly consider job durations. As pointed out in De Reyck and Leus (2008), when cash flows (costs and payoff) are not discounted, or more generally when the cash flows are time-independent, then it is a dominant decision to not schedule jobs in parallel. A second motivation for restricting the analysis to sequential schedules only is the possible use of a scarce (bottleneck) resource that can perform only one job at a time, e.g., expensive equipment or highly skilled labor. Therefore, a *schedule* $\mathbf{s}$ is an ordered subset of the activities of a scheduling instance. By s_t we denote the job in position t. A schedule $\mathbf{s}$ is *feasible* for scenario $\mathbf{x}$ if all precedence constraints are satisfied, both within modules and between modules. That is, a job in a feasible schedule can only be scheduled when all the preceding modules have been successfully completed (which depends on the scenario $\mathbf{x}$) and all preceding jobs in the same module have been attempted (which does not depend on the scenario $\mathbf{x}$). The set of schedules that is feasible under scenario $\mathbf{x}$ is denoted by $\Sigma_{\mathbf{x}}$. The set of all feasible schedules is given by $\Sigma = \bigcup_{\mathbf{x}\in\mathbb{B}^n} \Sigma_{\mathbf{x}}$.

To clarify the notion of feasibility and how it depends on the scenario, consider the following simplified story for the development of a new pharmaceutical substance into a drug. The precedence constraints in this story are represented graphically in Fig. 1. We will look at this situation from the perspective of the scientist who discovered a new substance called 'D'. The scientist has to exert effort to complete the jobs, and if the process ends with a viable drug then she receives a financial incentive (which is irrespective of the drug's market size). Module 1 represents the toxicity test. There are two jobs in this module: job 1 is preparing a batch of D and testing its toxicity, and job 2 is mixing some of the remaining D with additional substances and then testing the toxicity. Module 2, which contains only job 3, represents seeking budget approval from the firm's head of R&D to run clinical trials. Notice that in the precedence network in Fig. 1, there are no precedence relations between modules 1 and 2, i.e., the scientist can ask for budget approval for clinical trials before or after she has run the toxicity test. In practice, even though the order between modules 1 and 2 is unconstrained, she will likely first run the toxicity tests. The reason is that one can expect the ratio of cost to probability of failure of module 1 to be lower, since the toxicity test seems a lot more likely to fail (we refer to Sect. 3 for a formal derivation of the correctness of such ordering based on the ratios). It could also be the case, however, that the scientist's boss is very stringent on budget approvals, or that

the toxicity tests require a lot of effort. In that case, she would probably be better off first seeking budget approval for clinical trials before making the effort of running the toxicity tests. Module three represents clinical trials for effectiveness against medical condition A (job 4) and for condition B (job 5).

Now consider scenario $\mathbf{x}_1 = (0,1,1,0,1)$: activities 2, 3 and 5 succeed, but 1 and 4 fail. In the story, this scenario implies that pure D is toxic, but mixed D is not. The scientist will receive budget approval if she seeks it, and the drug based on mixed D will prove effective for condition B but not for condition A. Schedules $\mathbf{s}_1 = (1,2,3,5)$ and $\mathbf{s}_2 = (1,2,3,4)$ are examples of feasible schedules for scenario $\mathbf{x}_1$. Only the first schedule will be successful, since it tests for effectiveness against condition B. Schedules $\mathbf{s}_3 = (2,3,5)$ and $\mathbf{s}_4 = (1,3,5)$, on the other hand, are not feasible for scenario $\mathbf{x}_1$. Clearly, $\mathbf{s}_3$ is never feasible because the precedence constraint $(1,2) \in B_1$ requires activity 1 to be scheduled before activity 2; this infeasibility is independent of the scenario. In the story, one simply cannot test the mixed version of D before a batch of the pure version has been prepared for the initial test. Schedule $\mathbf{s}_4$ is infeasible for $\mathbf{x}_1$ since activity 5 of module 3 can only be started when all preceding modules were first completed successfully, and this is not the case for module 1 (since job 1 fails in scenario $\mathbf{x}_1$ and is the only job of module 1 in $\mathbf{s}_4$). In the story, the scientist is not allowed to test for effectiveness before she has proven that the substance is non-toxic in either its pure or mixed form. Under scenario $\mathbf{x}_1$, schedule 4 is not feasible, i.e., it can never be the outcome of the project. Contrast this with schedule $\mathbf{s}_5 = (1,3)$. This schedule will not be successful, yet it is feasible, since it respects all precedence constraints. To see how feasibility can depend on the scenario, now consider scenario $\mathbf{x}_2 = (1,1,1,1,1)$. Under this scenario, $\mathbf{s}_4$ becomes both feasible and successful, whereas $\mathbf{s}_5$ remains feasible but would still be unsuccessful.

2.2 Problem statement

The problem under study is to maximize the expected profit of a modular project by deciding which jobs to schedule and in which order, taking into account the precedence constraints. The payoff is only obtained when the project is completed successfully. The project success function for our setting is defined as:

$$\sigma(\mathbf{x},\mathbf{s}) = \begin{cases} 1 & \text{if } \mathbf{s} \text{ is successful for } \mathbf{x}, \\ 0 & \text{otherwise,} \end{cases}$$

where $\mathbf{s}$ is a schedule that is feasible for $\mathbf{x}$. We define $\mathbf{s}$ to be successful for scenario $\mathbf{x}$ if for every module there is at least one job scheduled that turns out to be successful, i.e.,

$$\forall i \in M : \exists s_t \in N_i \text{ s.t. } x_{s_t} = 1.$$

For a scenario $\mathbf{x}$ and a non-empty feasible schedule $\mathbf{s}$, the project's profit is given by

$$F(\mathbf{x},\mathbf{s}) = \sigma(\mathbf{x},\mathbf{s})V - \sum_{i=1}^{|s|} c_{s_i}. \tag{1}$$

When the schedule is empty ($\mathbf{s} = \emptyset$) we set $F(\mathbf{x}, \mathbf{s}) = 0$.

Reconsider the example instance and the feasible schedules $\mathbf{s}_1 = (1,2,3,5)$ and $\mathbf{s}_2 = (1,2,3,4)$ for scenario $\mathbf{x}_1 = (0,1,1,0,1)$. For this scenario, $\mathbf{s}_1$ is successful, but $\mathbf{s}_2$ is not. When all $c_i = 1$, $i = 1,\ldots,5$, and $V = 5$, we have $F(\mathbf{x}_1, \mathbf{s}_1) = 1$, whereas $F(\mathbf{x}_1, \mathbf{s}_2) = -4$ (a negative profit of -4, or loss of 4).

Due to the inherent uncertainty, a solution to this scheduling problem is not a single schedule but rather a scheduling *policy*, which is a set of rules that dynamically decide which scheduling decisions to make (which new jobs to start) based on the observed history of the project. A policy Π can be seen as a function $\Pi : \mathbb{B}^n \to \Sigma$, mapping scenarios $\mathbf{x}$ to schedules $\mathbf{s} = \Pi(\mathbf{x})$ that are feasible with respect to $\mathbf{x}$. Alternatively, a policy can be represented by a binary decision tree, where each node represents an activity and each pair of branches emanating from a node represents failure (left branch) or success (right branch) of the corresponding activity; the root node of the tree decides on the first job to schedule. In this way, the schedule $\mathbf{s} = \Pi(\mathbf{x})$ for an arbitrary scenario $\mathbf{x}$ corresponds to a unique path in the binary decision tree from the root node down to a leaf node.

An example of a scheduling policy Π for the project with precedence network as in Fig. 1 is given in Fig. 2. The leaf nodes are marked 'F' for failure or 'S' for success. This policy starts with job 3 of module 2. If job 3 fails, module 2 fails and the project fails as well. Note that this means that for scenarios of the form $\mathbf{x} = (\cdot,\cdot,0,\cdot,\cdot)$, policy Π will always produce the schedule $\mathbf{s} = (3)$. Under scenarios of this form, scheduling jobs 4 or 5 after job 3 would not be feasible given the precedence constraints. Scheduling job 1 would be allowed: $\mathbf{s} = (3,1)$ would be feasible under $\mathbf{x} = (\cdot,\cdot,0,\cdot,\cdot)$, but it would never be successful given the failure of job 3, which is why this policy wisely chooses not to attempt it. If job 3 succeeds, job 1 is scheduled. If this job succeeds, module 1 is completed and module 3 can be executed. For module 3, the policy schedules job 4 if the outcome of job 5 is unsuccessful. In other words, for scenarios $(1,\cdot,1,\cdot,0)$ this policy will produce the schedule $\mathbf{s} = (3,1,5,4)$. For module 1, this policy will never schedule job 2, so if job 1 fails, module 1 fails and the entire project is abandoned (which means no more activities are scheduled, and the payoff is not obtained). In other words, the issue of job *selection* is inherent in the problem statement.

Problem MP1 ('Modular Project scheduling on One Machine') can now be formally stated as follows: from a given class of policies $\mathscr{C}$, select a policy Π^* that maximizes the expected profit of the project:

$$\Pi^* = \underset{\Pi \in \mathscr{C}}{\arg\max}\, \mathbb{E}\Big[F\big(\mathbf{X}, \Pi(\mathbf{X})\big)\Big],$$

with $\mathbb{E}[\cdot]$ denoting the expectation operator with respect to the random variable $\mathbf{X}$. This problem has been proved to be NP-hard, even for certain restrictive special cases (Coolen et al. 2014). This means that, unless P = NP, no exact polynomial-time algorithm exists, which motivates our search for an efficient heuristic that provides good solutions to MP1 instances in polynomial time. We assume that the 'empty policy,' which corresponds to an abandonment of the project immediately when it is started, is an element of any policy class. Therefore, any policy with a negative expected profit is dominated by the empty policy, which has zero expected profit.

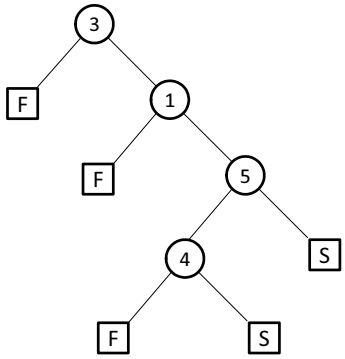


Fig. 2 A scheduling policy for the project in Fig. 1

We note that problem MP1 can alternatively be described as a Markov decision process (Puterman 1994). The system state at a given decision epoch corresponds to the current progress of the project and is completely determined by the subset of jobs that are still idle at that time, where a job is said to be idle if it has not yet been started and success is not yet achieved for its module. The allowable actions in a state at a given decision epoch are twofold: either the decision maker decides to abandon the project, leading to a zero reward, or he decides to execute an eligible job from the set of remaining jobs (one that is available according to the precedence constraints). In the latter case, the decision maker receives a negative reward equal to the cost of that job, unless it is the dummy end job, in which case a positive reward is incurred corresponding with the project payoff. Finally, the transitions from a given state and selected action at a decision epoch to the state at the next decision epoch are determined by the success and failure probabilities of the jobs. Coolen et al. (2014) propose a backward stochastic dynamic-programming algorithm with state space as described in this paragraph. The value function can be computed recursively by choosing at each state the best allowable action based on the best allowable actions determined at previously computed states in the dynamic-programming algorithm.

2.3 Classes of scheduling policies

Various classes of scheduling policies are presented in Coolen et al. (2014); the definitions below mainly draw from that source. We will restrict our solution space to only one class, namely the 'elementary module-sequence policies.'

We first describe the class $\mathscr{C}_E$ of *elementary* policies. An elementary policy is characterized by a compatible ordering L of a subset of N. A compatible ordering is an ordering that respects all precedence constraints, which means that L should be a linear extension of the subgraph of (N, B^*) induced by L. In this definition, it is assumed that $L = \emptyset$ is also possible, which corresponds to the empty policy. From this ordering L and for every scenario, a schedule is generated as follows: start with

the first job in L; if it fails and there are no more jobs of its module left in L, abandon the project. If the job succeeds, this means the module is completed and all other jobs from this module can be deleted from L. Iterating through L in this way, the schedule is then simply given by the jobs that were considered.

Another class of interest to us is that of the *module-wise* policies $\mathscr{C}_M$. A module-wise policy is a policy that only produces schedules in which all jobs belonging to the same module are executed consecutively, or informally: there is no 'jumping' between modules. A *module-sequence* policy then, is a module-wise policy that adheres to the same linear extension of the module order A for each possible realization. Finally, policies that are both elementary and module-sequence are called *elementary module-sequence* policies or *EMS*. The class of EMS policies is denoted by $\mathscr{C}_{EMS}$. The policy depicted in Fig. 2, for example, is an EMS policy, characterized by the ordering $L = (3,1,5,4)$.

Based on a number of desirable properties of the class of elementary module-sequence policies established in Coolen et al. (2014), we restrict our search to EMS policies only: first, they have a compact representation – their ordering L – that is easy to interpret and work with; second, their expected profit can be evaluated in linear time; and third, there always exists an EMS policy that is optimal in the larger class $\mathscr{C}_E \cup \mathscr{C}_M$, even though EMS policies are less in number. There is, however, an important caveat: EMS policies are not *globally optimal*. This means that there exist MP1 instances for which no elementary policy is optimal. EMS policies can even turn out to be arbitrarily bad vis-à-vis the global optimum; more precisely, there are instances for which an optimal EMS policy is defined by the empty list (with zero profit) whereas binary decision trees defining non-elementary policies with strictly positive expected profit exist (see Coolen et al. (2014) for such an instance). Even though the worst-case performance of an optimal EMS policy may be arbitrarily bad, the average performance over a large dataset of 'typical' instances turns out to be quite good: in Coolen et al. (2014), the average difference between globally optimal policies and optimal elementary policies was below 0.01% (less than one hundredth of a percent). Unfortunately, finding an optimal elementary policy has turned out to be quite time-consuming, which is why in this text we focus on finding good heuristic EMS policies.

3 Sequential testing problems

In this section, we review a number of concepts and results from the sequential testing literature that can be transposed to the MP1 setting. The heuristic for MP1, to be described in Sect. 4, is based on an optimal procedure for testing so-called 'series-parallel' systems.

3.1 Link between testing and scheduling

A review of the literature on sequential testing can be found in Ünlüyurt (2004). In testing problems, one is concerned with the system function f rather than with

the Boolean project success function as used in MP1, for a system consisting of a number of components. The system function f indicates the state of the system for every possible combination of states of the underlying components. The state of the components is given by a vector $\mathbf{x}$, with $x_k = 1$ if component k is working, and $x_k = 0$ if it is failing – we deliberately re-define some symbols, to underline their equivalence in the two settings (scheduling and testing); obviously, we equate activities in the scheduling problem with components in the testing problem. When the system is working, $f(\mathbf{x}) = 1$. When it is failing, $f(\mathbf{x}) = 0$. Each of the state variables x_k is the outcome of a Bernoulli variable X_k with probability of success p_k. The probability that the component fails is given by $q_k = 1 - p_k$. The probabilities p_k are gathered in vector $\mathbf{p}$ (with p_k as its k-th component).

The Boolean system function that has the closest ties to our MP1 scheduling problem is given by:

$$f(\mathbf{x}) = \bigwedge_{i \in M} \left(\bigvee_{k \in N_i} x_k \right), \tag{2}$$

where $\wedge$ denotes the Boolean AND-function and $\vee$ the Boolean OR-function. This system function corresponds to a *series-parallel system of depth 2*. A *series-parallel system* (SPS) can be defined recursively as a system that consists of a serial or parallel connection of subsystems that are themselves SPS (Ünlüyurt 2004). The depth of an SPS is defined as $1 + \max\{\text{depth of proper subsystem}\}$. For example, (simple) serial and parallel systems can be thought of as SPSs of depth 1, with system functions $f(\mathbf{x}) = \wedge_i x_i$ and $f(\mathbf{x}) = \vee_i x_i$, respectively. There are two types of SPSs of depth 2, which are named according to their global structure. A two-level SPS that is a serial connection (respectively parallel arrangement) of parallel (respectively serial) subsystems, is also simply called a series-parallel (respectively parallel-series) system (Ben-Dov 1981b).

A solution to a testing instance is also a policy Π, which maps each state vector $\mathbf{x} \in \mathbb{B}^n$ to a schedule $\mathbf{s}$. A major difference between testing and scheduling, however, is the following: when testing, one will continue exploring new components until the actual state of the system is known with certainty. This means that, regardless of the order in which components are tested, the final output will always be the actual state of the system. In MP1, on the other hand, one can abandon the project prematurely, the result being that the project fails even though it may have been possible to complete it successfully had more activities been undertaken, but the remaining activities may simply be too costly compared to the expected payoff. As a consequence, a scheduling policy Π may fail to identify $f(\mathbf{x})$ for certain scenarios $\mathbf{x} \in \mathbb{B}^n$, so that $\sigma(\mathbf{x}, \Pi(\mathbf{x})) = 0$ even though $f(\mathbf{x})$ may have been equal to 1.

The second difference between testing and scheduling is the objective function. When testing, the goal is to minimize the expected cost of discovering the state of the system. The cost function $G(\mathbf{s})$ for schedule $\mathbf{s}$ is given by $\sum_{i \in \mathbf{s}} c_i$. Depending on the context, this can represent a financial cost or the time needed to test the components, or another measure of testing effort. A generic statement of the sequential testing problem, in line with our problem statement in Sect. 2.2, is then:

$$\Pi^* = \operatorname*{arg\,min}_{\Pi \in \mathscr{T}} \mathbb{E}\Big[G\big(\Pi(\mathbf{X})\big) \Big], \tag{3}$$

where $\mathscr{T}$ is a given class of testing policies. When no precedence constraints apply to the sequencing of the component tests and with a system function of the form (2), we have the unconstrained series-parallel sequential testing problem (USPST).

The concept of an elementary policy can also be defined for testing problems. In light of the impossibility of early abandonment (discussed supra), however, the compatible ordering L defining an elementary policy now needs to be an ordering of the entire component set N. One minor modification can be made, namely that L needs to contain only the non-redundant components of the instance, where a component k is redundant if it has no influence on the system, that is $\forall \mathbf{x} \in \mathbb{B}^n$ with $x_k = 0$ we have $f(\mathbf{x}) = f(\mathbf{x} \vee e_k)$, with e_k the k-th unit vector of $\mathbb{R}^n$. Without loss of generality, we will simply work with an ordering of N below. We thus conclude that the class $\mathscr{C}_E$ of elementary scheduling policies contains the class $\mathscr{T}_E$ of elementary testing policies, for the same parameters.

3.2 Optimality results for specific sequential testing problems

In this section, we discuss a number of sequential testing problems without precedence constraints for which elementary policies are globally optimal. For a simple series system, the order $(1,2,\ldots,n)$ is optimal if and only if $\frac{c_1}{q_1} \leq \frac{c_2}{q_2} \leq \ldots \leq \frac{c_n}{q_n}$. The proof relies on a straightforward interchange argument, and has been provided by Mitten in 1960 and by Butterworth in 1972; for discussions of this result, see for example Ben-Dov (1981b) and Ünlüyurt (2004). This result can be transferred to MP1: an elementary policy defined by the same ordering is optimal for an MP1 instance with each $|N_i| = 1$ and no precedence constraints (at least if V is large enough, otherwise the empty list is optimal).

Similarly, for a simple parallel system the order $(1,2,\ldots,n)$ is optimal if and only if $\frac{c_1}{p_1} \leq \frac{c_2}{p_2} \leq \ldots \leq \frac{c_n}{p_n}$. This ordering also defines an optimal elementary policy for MP1 instances with $|M| = 1$ and $B_1 = \emptyset$, with the caveat that jobs k satisfying

$$c_k > p_k V \tag{4}$$

should be removed (Coolen et al. 2014).

The optimal testing procedure for a series-parallel system can be derived using a two-stage approach based on the results above, following Ben-Dov (1981b). First, order the components within each parallel subsystem i (the testing equivalent of a module) according to the optimal testing procedure for parallel systems; denote the obtained order by L_i. Subsystem i as a whole then, has an associated probability of failure $\theta_i = \prod_{k \in N_i} q_k$ and an associated expected cost

$$\kappa_i(L_i) = \sum_{k=1}^{|L_i|} \Big(\prod_{l=1}^{k-1} q_{[l]_{L_i}}\Big) c_{[k]_{L_i}},$$

where empty products are taken to be 1. The symbol $[k]_\alpha$ denotes the object in the k-th position according to permutation α. When it is clear from the context, the permutation to be used (in this case L_i) will be omitted. The probability of success for subsystem i is denoted by $\pi_i = 1 - \theta_i$.

Second, order the subsystems (modules) according to the optimal testing procedure for series systems, so in non-decreasing order of $\frac{\kappa_i}{\theta_i}$. Denote the obtained order by σ; the optimal order is then given by $(L_{[1]_\sigma}, L_{[2]_\sigma}, \ldots, L_{[m]_\sigma})$. Testing of a subsystem stops as soon as one working component has been found; testing of the system halts as soon as one subsystem has failed or all subsystems are known to be successful. Only in the latter case, does the overall system function.

3.3 Reliability importance

An interesting concept from the testing literature is the reliability importance of a component, first defined by Birnbaum (1969) and used in exact testing algorithms, such as the one by Ben-Dov (1981a), and in approximation testing algorithms, such as by Jędrzejowicz (1983). In Sect. 4.2, we use this concept in the development of a heuristic procedure for MP1.

Let the *reliability function* h for a series-parallel system with system function as in Eq. (2) be the probability that the system will be working contingent on the component probabilities $\mathbf{p}$, so

$$h(\mathbf{p}) = \mathbb{E}[f(\mathbf{X})] = \prod_{i \in M} \left(1 - \prod_{k \in N_i} q_k\right) = \prod_{i \in M} \pi_i.$$

Following Birnbaum (1969), we define the *reliability importance* I_k of component k as the partial derivative of h with respect to p_k; this is a measure for the component's importance in determining whether the system will be up or down. If I_k is zero then the component is irrelevant. The larger I_k, the more crucial or relevant the component to overall system success. Assuming $k \in N_j$, we obtain:

$$I_k(\mathbf{p}) \equiv \frac{\partial h}{\partial p_k}(\mathbf{p}) = -\frac{\partial h}{\partial q_k}(\mathbf{p}) = \prod_{\substack{i \in M \\ i \neq j}} \left(1 - \prod_{l \in N_i} q_l\right) \prod_{\substack{l \in N_j \\ l \neq k}} q_l = \prod_{\substack{i \in M \\ i \neq j}} \pi_i \prod_{\substack{l \in N_j \\ l \neq k}} q_l. \qquad (5)$$

We conclude that a job will be more important when, relatively to the other modules, its module has a low probability of success, and when compared to the other jobs in its module, the job itself has a high probability of success. This is intuitive: since all modules have to succeed, modules with low success probability are more important. Conversely, since within a module the success of one job is sufficient, jobs with high success probability are more important.

4 A greedy heuristic

The optimal two-stage procedure for USPST is not directly applicable to MP1 because of two major differences. First, the MP1 setting includes precedence constraints, so the ordering produced by the USPST algorithm may not be feasible. Second, the list defining an elementary testing policy contains all jobs, while job selection may be beneficial to the MP1 objective. In this section, we propose three

different greedy heuristics for MP1, in increasing order of quality of the solutions produced. For ease of presentation and understanding, we present the algorithms sequentially from simplest to most complicated: each algorithm is an improvement of the previous one. The described procedures for MP1 are inspired by the exact algorithm for USPST, and we will consider EMS policies only, which can be conveniently represented by their compatible order list. Algorithm Greedy 1 generates an initial list of all jobs. Greedy 2 seeks to improve this initial solution by removing jobs from the list. In Greedy 3 we incorporate a randomization step to find better module orderings.

4.1 Greedy 1: An initial list

In this section, we describe a simple heuristic (referred to as Greedy 1) that generates an EMS policy for MP1 starting from the solution for the related USPST problem. In order to satisfy the precedence constraints between jobs inside each module as well as between the modules, a reordering of the jobs and the modules in the USPST solution is necessary. The first stage of the procedure of Ben-Dov (1981b) provides an ordering of the jobs L_i for each module i based on the ratio cost / success probability. Next, we iteratively build a feasible job ordering L_i' for each module i as follows. At each step, we greedily select the first unscheduled 'available' job appearing in L_i to be the next job in L_i', where a job is available if it has either no predecessors or if all its predecessors are already in L_i'. Next, we can apply the second stage of Ben-Dov's procedure: we compute for each module i a cost κ_i (based on the list L_i') and a failure probability θ_i and compute a module ordering σ_0 based on the ratios κ_i/θ_i. Finally, we apply the same greedy procedure that transforms the module ordering σ_0 into an ordering σ satisfying the module precedence constraints. The greedy subroutine that is used in Greedy 1 to transform a given job or module order into one that satisfies the precedence constraints, is described in detail in Algorithm 1; it transforms a given permutation into a linear extension of a partially ordered set (poset). Notice that in the computation of the set of eligible activities $\mathscr{E}$ (lines 2 and 6 of Algorithm 1), we only need to consider the immediate predecessors. Therefore, we use the transitive reduction[2] of (Z,E) in the implementation of Algorithm 1.

To illustrate the functioning of this procedure, we consider the module depicted in Fig. 3. Assume that in the absence of the precedence constraints, an optimal order within this module is $L_{i0} = (3,2,4,5,1)$ (ordered in non-decreasing $\frac{c_i}{p_i}$). Greedy 1 then calls Algorithm 1 to transform L_{i0} into $L_i = (2,4,5,1,3)$.

4.2 Greedy 2: Deciding which jobs not to schedule

In order to decide which jobs not to include, we will compare the expected incremental benefit of scheduling a job with the expected incremental cost. The expected payoff when scheduling all jobs is $EP = h(\mathbf{p})V$. Not scheduling job k is equivalent to setting its p_k equal to 0. Denote by $\mathbf{p}^{(k)}$ the vector $\mathbf{p}$ with a zero at component k,

[2] To compute the transitive reduction of a poset (V,O) we remove all elements (i,l) of O for which there is an element $j \in V$ such that $(i,j) \in O$ and $(j,l) \in O$.

Greedy 1

Input: MP1 instance
Output: List L defining an EMS policy

1: **for all** $i \in M$ **do**
2: Sort elements k of N_i in non-decreasing order of $\frac{c_k}{p_k}$, put the result in L_{i0};
3: $L_i \leftarrow$ Algorithm1$((N_i, B_i), L_{i0})$;
4: Compute $\kappa_i(L_i)$ and θ_i;
5: **end for**
6: Sort elements i of M in non-decreasing order of ratio $\frac{\kappa_i(L_i)}{\theta_i}$, put the result in σ_0;
7: $\sigma \leftarrow$ Algorithm1$((M, A), \sigma_0)$;
8: Set $L \leftarrow (L_{[1]_\sigma}, \ldots, L_{[m]_\sigma})$;
9: If the EMS policy defined by L has a negative expected profit, we set $L = \emptyset$;
10: Return L;

Algorithm 1 Basic subroutine to produce a linear extension of a poset

Input: poset (Z, E) and permutation L_0 of Z
Output: Algorithm1$((Z, E), L_0) \equiv$ linear extension L of (Z, E)

1: Initialize $L = \emptyset$;
2: Initialize $\mathscr{E} = \{i \in L_0 : \text{ for all } (j,i) \in E, j \in L\}$; {$\mathscr{E}$ is the list of 'available' jobs}
3: **while** $L_0 \neq \emptyset$ **do**
4: Let j^* be the first element of L_0 that belongs to $\mathscr{E}$;
5: Append j^* to L and remove it from L_0;
6: Update $\mathscr{E}$;
7: **end while**
8: **return** L

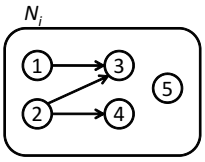


Fig. 3 A module with precedence constraints between jobs

i.e., $\mathbf{p}^{(k)} = (p_1, \ldots, p_{k-1}, 0, p_{k+1}, \ldots, p_n)$. We derive the following first-order Taylor approximation for $h(\mathbf{p})$ around the point $\mathbf{p}^{(k)}$:

$$h(\mathbf{p}) \approx h(\mathbf{p}^{(k)}) + 0 \cdot \frac{\partial h}{\partial p_1}(\mathbf{p}^{(k)}) + \cdots + p_k \cdot \frac{\partial h}{\partial p_k}(\mathbf{p}^{(k)}) + \cdot + 0 \cdot \frac{\partial h}{\partial p_n}(\mathbf{p}^{(k)}). \quad (6)$$

By Eq. (5), $\frac{\partial h}{\partial p_k}(\mathbf{p}^{(k)}) = I_k(\mathbf{p}^{(k)}) = I_k(\mathbf{p})$, and so the incremental change in expected payoff ΔEP from removing job $k \in N_j$ is given by:

$$\Delta EP = h(\mathbf{p}^{(k)})V - h(\mathbf{p})V = -p_k I_k(\mathbf{p})V = -\left(\prod_{\substack{i \in M \\ i \neq j}} \pi_i \prod_{\substack{l \in N_j \\ l \neq k}} q_l \right) p_k V.$$

This result can be stated as an equality, since h is linear in each of its arguments, so the first-order approximation of Eq. (6) actually holds with equality. Note that this

derivation ignores within-module precedence relationships. To maintain a feasible solution, only jobs without successor jobs in that module can be removed.

We denote the expected cost by EC. Contrary to EP, the value of EC depends on the order list at hand. We derive the incremental change compared to an initial list $L = (L_{[1]_\sigma}, \ldots, L_{[m]_\sigma})$ of all jobs, as generated by Greedy 1 for instance. For a module j, the incremental change in expected cost ΔEC from removing the last job k from L_j is:

$$\Delta EC = -\left(\prod_{i \in F(j)} \pi_i \prod_{\substack{l \in N_j \\ l \neq k}} q_l\right)\left(c_k + p_k \sum_{i \in G(j)} \left(\prod_{h \in G(j) \cap F(i)} \pi_h\right) \kappa_i\right),$$

where $F(j)$ is the set of modules scheduled before module j and $G(j)$ is the set of modules scheduled after j. Note that $\{F(j), \{j\}, G(j)\}$ is a partition of M, for any $j \in M$. The formula for ΔEC only holds for evaluating the impact of the removal of a job that is scheduled last in the module. A generalization for removal of a job that is not last in L_j is possible, but depends on the jobs scheduled after job k in L_j. Such a generalization is not incorporated in the selection rule derived below because it is only valid for jobs without successor jobs in that module.

In conclusion, the net incremental expected profit of the removal of job k that was scheduled last in a module j, is given by:

$$\Delta EP - \Delta EC = \left(\prod_{i \in F(j)} \pi_i \prod_{\substack{l \in N_j \\ l \neq k}} q_l\right)(c_k + p_k \Gamma_j) - \left(\prod_{\substack{i \in M \\ i \neq j}} \pi_i \prod_{\substack{l \in N_j \\ l \neq k}} q_l\right) p_k V, \qquad (7)$$

where

$$\Gamma_j = \sum_{i \in G(j)} \left(\prod_{h \in G(j) \cap F(i)} \pi_h\right) \kappa_i \qquad (8)$$

denotes the expected cost of the project after module j is completed. If all modules before j are successful, and all jobs in N_j have failed, then if we are willing to forfeit the extra attempt offered by job k, not only will we save its own cost c_k, but if job k was successful, the expected cost of all modules further on in the schedule would also no longer have to be laid out.

It seems preferable not to schedule a job when the net incremental expected profit given by Eq. (7) is positive. In the heuristics that are proposed in the paper, we decide to apply this selection criterion for any job in a module in order to evaluate the desirability of including the job in a schedule, even though the rule is designed only for jobs that are scheduled last in a module. We do this because only removing jobs without successors will not lead to very good solutions if there are very undesirable jobs present that have successor jobs. This rule should be conservative in the sense that we are more likely to keep a job scheduled than in the apparent optimum. Given the precedence constraints that have to be accounted for it is probably better to be conservative. Indeed, when a job is removed from the order list, all its successors

need to be removed as well. Since this effect is difficult to factor in explicitly (and we did not in fact incorporate it in the derivations above), for a heuristic rule it seems better to be conservative about removing a job from the list. To derive a conservative heuristic rule we underestimate $\Delta EP - \Delta EC$.

Our negatively biased simplification of Eq. (7) consists in working with a smaller likelihood of saving the extra costs of the project from module j onwards. To simplify notation, let

$$\prod_{\substack{i\in M\\ i\neq j}} \pi_i = \alpha_j,\ \prod_{i\in F(j)} \pi_i = \beta_j \text{ and } \prod_{i\in G(j)} \pi_i = \gamma_j,$$

with an empty index set resulting in a product of 1. It holds that $\alpha_j \leq \beta_j$, $\alpha_j = \beta_j\gamma_j$ and

$$\begin{aligned}\Delta EP - \Delta EC &= \beta_j \left(\prod_{\substack{l\in N_j\\ l\neq k}} q_l\right)(c_k + p_k\Gamma_j) - \alpha_j \left(\prod_{\substack{l\in N_j\\ l\neq k}} q_l\right) p_k V \\ &\geq \beta_j \left(\prod_{\substack{l\in N_j\\ l\neq k}} q_l\right) c_k - \alpha_j \left(\prod_{\substack{l\in N_j\\ l\neq k}} q_l\right) p_k(V-\Gamma_j).\end{aligned}$$

We anticipate that it is better not to have job k scheduled at the end of module j if the right-hand side of the latter inequality (which is an underestimation of Eq. (7)) exceeds zero, so if

$$\beta_j c_k \geq \alpha_j p_k (V - \Gamma_j).$$

With $\alpha_j = \beta_j\gamma_j$, we obtain the following heuristic selection rule:

$$\frac{c_k}{p_k} \geq \gamma_j (V - \Gamma_j). \tag{9}$$

That is, remove job $k \in N_j$ from the list if $\frac{c_k}{p_k}$ exceeds a fraction γ_j of the difference between the project payoff V and the expected further project cost Γ_j. Note that for instances with $|M| = 1$ and $B_1 = \emptyset$, this is exactly the optimal selection rule described by Condition (4). In the heuristics that are proposed in the paper, we decide to use this rule for any job in a module in order to evaluate the desirability of including the job in a schedule, even though the rule is designed only for jobs that are scheduled last in a module. Only removing jobs without successors will not lead to very good solutions if there are very undesirable jobs present that have successor jobs. Moreover, we do not wish to use an iterative approach to remove jobs together with their successors, as this would require dynamically recomputing the selection rule to deal with successor jobs that have been removed in previous iterations. We opt for a static selection rule (only computed once) that is efficient, and easy to understand and interpret in terms of the characteristics of the job.

In Greedy 2 we start from the list L returned by Greedy 1, which contains all the jobs. Then, for each module, we remove all jobs that satisfy Eq. (9) from L together with their successors, resulting in a shorter list L'. Successor jobs must be removed as

Greedy 2

Input: MP1 instance
Output: List defining an EMS policy

Apply Greedy 1 to obtain a list $L = (L_{[1]}, L_{[2]}, \ldots, L_{[m]})$;
for all $i \in M$ **do**
  Compute γ_i and Γ_i;
  Let k be the smallest index of L_i such that $\frac{c_{[k]}}{p_{[k]}} \geq \gamma_i(V - \Gamma_i)$;
  if no such index exists **then**
    Set $k = |L_i| + 1$; {to ensure all jobs are scheduled if none exceeds the critical value}
  else if $k = 1$ **then**
    Set $k = 2$; {to ensure every module keeps at least one job}
  end if
  Set $L'_i \leftarrow ([1]_{L_i}, [2]_{L_i}, \ldots, [k-1]_{L_i})$;
  Compute $\kappa_i(L'_i)$ and $\theta_i(L'_i) = \prod_{k \in L'_i} q_k$;
end for
Set $L' \leftarrow (L'_{[1]}, \ldots, L'_{[m]})$;
Sort elements $i \in M$ in non-decreasing order of $\kappa_i(L'_i)/\theta_i(L'_i)$ and put the result in σ'_0;
$\sigma' \leftarrow$ Algorithm1$((M,A), \sigma'_0)$;
Set $L'' \leftarrow (L'_{[1]_{\sigma'}}, \ldots, L'_{[m]_{\sigma'}})$;
return L, L' or L'' – pick one with the highest expected profit;

well to satisfy the precedence constraints. Removing jobs from a module i increases θ_i and decreases κ_i, and thus module i will have a lower ratio κ_i/θ_i. Therefore, we may reorder the modules in L' by applying Algorithm 1 to the updated module order of non-decreasing ratio κ_i/θ_i, resulting in another list L''; this latter list, however, does not necessarily lead to a better policy than the list L'. There is also no guarantee that the selection process will improve the initial list L. Greedy 2 returns from the three lists L, L' and L'', one with the highest expected profit.

Table 1 A module with jobs that are not retained by Greedy 2

k	c_k	p_k	c_k/p_k
1	46	0.961	47.9
2	10	0.891	11.2
3	2	0.895	2.2
4	12	0.836	14.4
5	41	0.912	45.0

Consider again the module depicted in Fig. 3, for which Greedy 1 found the list $L_i = (2,4,5,1,3)$. This module is part of a larger test instance, and part of the actual data are reproduced in Table 1. Assume that $V = 122$, $\gamma_i \approx 0.983$ and $\Gamma_i \approx 93.0$ (these values can only be determined on the basis of the entire instance, which is reproduced in Appendix A), then we have a critical value $\gamma_i(V - \Gamma_i) \approx 28.4$. Two ratios exceed this threshold: $\frac{c_1}{p_1}$ and $\frac{c_5}{p_5}$; job $5 = [3]_{L_i}$ is the first job in L_i with a ratio exceeding the threshold. Note that, based on the functioning of Algorithm 1, the jobs after 5 in L_i either have a more unfavorable ratio (such as 1) or are successors of some job that

exceeds the threshold and must therefore be removed in order to obtain a feasible solution (job 3 is a successor of job 1). Consequently, the initial list $L_i = (2,4,5,1,3)$ is cut off at the job with the lowest ratio exceeding the threshold (job 5), leading to the order $L_i' = (2,4)$.

4.3 Greedy 3: Randomizing the module order

On studying the computational results of Greedy 1 and Greedy 2 (a detailed summary of these results is provided in Sect. 5), we observe that modules are frequently not in an optimal order because of the greediness of Algorithm 1. Consider the following example with three modules: $A = \{(1,2)\}$ and, for a certain order of their jobs, $\kappa_1 = 4$, $\kappa_2 = 1$, $\kappa_3 = 3$ and $\theta_1 = 0.2$, $\theta_2 = 0.5$, $\theta_3 = 0.2$. The ratios κ/θ are 20, 2 and 15, and the desired order $\sigma_0 = (2,3,1)$. Calling Algorithm1((M,A),σ_0) yields $\sigma = (3,1,2)$, with expected cost $\kappa_3 + \pi_3(\kappa_1 + \pi_1\kappa_2) = 6.84$. Even though module 2 is clearly preferable and has a low cost, it cannot be placed first because module 1 is a predecessor. Algorithm 1 places module 3 first because module 1 has a ratio that is slightly higher than that of module 3. It is an optimal decision, however, to select module 1 first in order to enable module 2 to come earlier. The order $(1,2,3)$ has (lower) expected cost $\kappa_1 + \pi_1(\kappa_2 + \pi_2\kappa_3) = 6$.

The previous example shows that greedy module orderings can be suboptimal. To overcome this, we have introduced a randomization step in order to increase the likelihood of finding good module orderings. Instead of always choosing from the set $\mathscr{E}$ of eligible modules one with lowest ratio κ/θ (see Algorithm 1), we now create the possibility of selecting a module with a higher ratio (Algorithm 2) by means of regret-based random sampling (RBRS) (Drexl 1991). This idea is also similar to the notion of stochastic ranking that was popularized by evolutionary algorithms (Runarsson and Yao 2000). The probability P_i of selection of a module i out of the set $\mathscr{E}$ is determined as follows:

$$P_i = \frac{(\rho_i + 1)^\alpha}{\sum_{j \in \mathscr{E}} (\rho_j + 1)^\alpha}, \tag{10}$$

with $\rho_i = [\arg\max_{j \in \mathscr{E}} \kappa_j/\theta_j]_{L_0} - [i]_{L_0}$ and where $[j]_{L_0}$ denotes the position of module j in the initial module ordering L_0 (non-decreasing κ/θ).

In Greedy 3, we start from the result of Greedy 2 to guarantee producing a solution that is at least as good. Subsequently, we invoke Greedy 2 for different module orderings obtained via Algorithm 2 until a stopping criterion is met. We only use Algorithm 2 to generate random module orderings before job selection. After job selection, i.e., at line 15 of Greedy 2, we still use the deterministic Algorithm 1. This is to avoid nesting randomizations.

We choose the values ρ_i of module i in terms of the position of module i in the module ordering L_0 rather than the ratio κ/θ itself, because the ratio can take rather extreme values (especially for large modules). The parameter $\alpha \in [0,\infty)$ controls the diversification of the sample of module orderings selected from the population of all possible module orderings. The diversification in the module lists decreases with α. The boundary case of $\alpha = 0$ corresponds to a completely random selection from $\mathscr{E}$ with equal probability $1/|\mathscr{E}|$, whereas in the other extreme of $\alpha = +\infty$, we always

Greedy 3

Input: MP1 instance
Output: List defining an EMS policy

1: Let L be the output of Greedy 2;
2: **while** Stop criterion not met **do**
3: Let $\bar{L}$ be the output of Greedy 2 with subroutine Algorithm 1 replaced by Algorithm 2 when line 7 of Greedy 1 is called;
4: Update L if $\bar{L}$ has higher expected profit;
5: **end while**
6: **return** L;

Algorithm 2 RBRS variant of Algorithm 1

Input: poset (Z,E) and permutation L_0 of Z
Output: Algorithm2$((Z,E),L_0) \equiv$ linear extension L of (Z,E)

1: Initialize $L = \emptyset$;
2: Initialize $\mathscr{E} = \{i \in L_0 : \text{ for all } (j,i) \in E, j \in L\}$;
3: **while** $L_0 \neq \emptyset$ **do**
4: Randomly pick one element $j^* \in \mathscr{E}$, where each $j \in \mathscr{E}$ has probability P_j of being selected according to Eq. (10);
5: Append j^* to L and remove it from L_0;
6: Update $\mathscr{E}$;
7: **end while**
8: **return** L

select that element of $\mathscr{E}$ appearing first in the initial ordering L_0 (as in Algorithm 1). In Sect. 5.1 we elaborate on the choice of the parameter α and on the stop criterion.

5 Computational experiments

In this section, we assess the performance of the algorithms on a dataset[3] of 360 instances. The instances are indexed according to their size, measured by the number n of jobs, and their order strength OS. The order strength is defined as the number of precedence-related activity pairs in the induced network (N,B^*) divided by the maximum possible number of such pairs, and therefore equals $|B^*|/\binom{n}{2}$. For each combination of $n \in \{10,20,30,\ldots,120\}$ and $OS \in \{0.4,0.6,0.8\}$, 10 instances were created. Full details on the data generation can be found in Coolen et al. (2014).

All experiments were run on a Dell Latitude D830 with a 2.5 GHz processor and 3 GB of RAM, running 32-bit Windows Vista. The algorithms were implemented in C++ using Microsoft Visual Studio 2010. We evaluate the performance of the greedy heuristics against the two exact algorithms developed in Coolen et al. (2014): a branch-and-bound (B&B) algorithm that finds an optimal EMS policy, and a dynamic-programming (DP) algorithm that outputs a globally optimal policy. To facilitate comparison, we define the relative optimality gap ROG of an algorithm $\mathscr{A}$ for an instance of MP1 as follows:

$$ROG = \frac{z^* - z(\mathscr{A})}{z^*} \text{ if } z^* \neq 0, \text{ and 0 otherwise,} \tag{11}$$

[3] Available online at `http://feb.kuleuven.be/public/NDBAC96/MP1_instances.htm`

where z^* is the objective value of a globally optimal solution (found by DP) and $z(\mathscr{A})$ the objective of the output of $\mathscr{A}$.

In Sect. 5.1, we include a discussion of some implementation choices for heuristic Greedy 3, followed by a presentation of the computational results in Sect. 5.2.

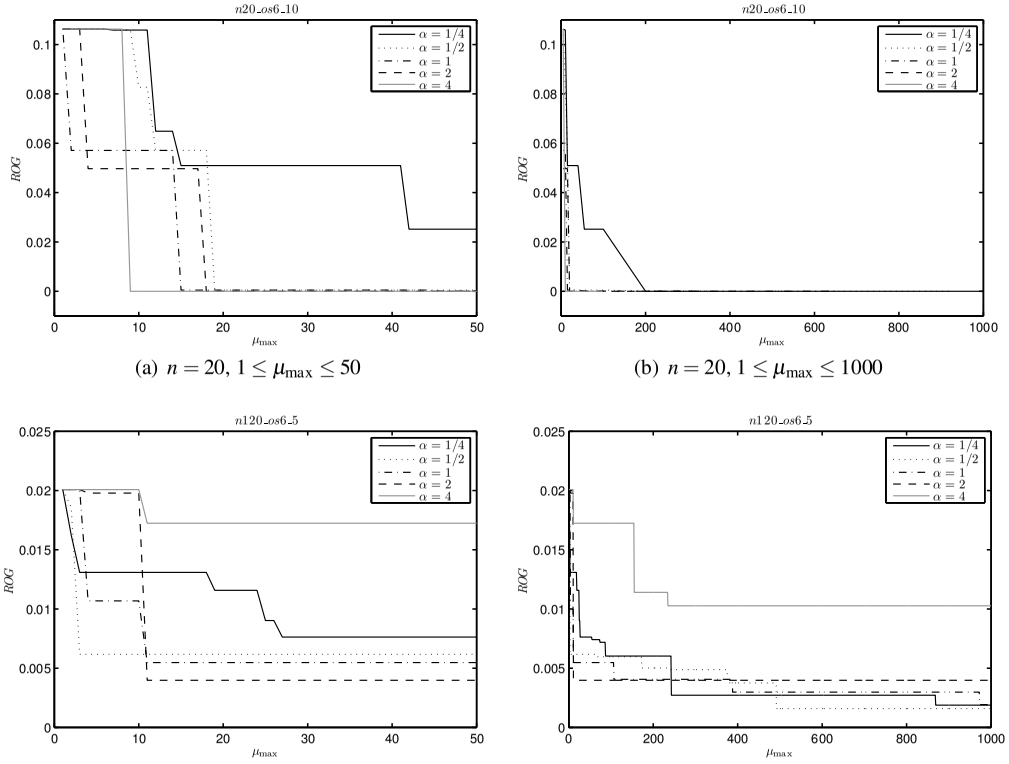
5.1 Implementation choices for Greedy 3

For the implementation of algorithm Greedy 3, we need to decide on the stop criterion and on the value of the parameter α. Reaching a maximum allowable number $\mu_{\max}$ of different module orders generated by Algorithm 2 is a natural stop criterion. In Fig. 4 we show the relative optimality gap as a function of $\mu_{\max}$ for different values of α, for a small instance with 20 jobs (Figs. 4(a) and 4(b)), and for a large instance with 120 jobs (Figs. 4(c) and 4(d)). Plots (a) and (c) focus on small values of $\mu_{\max}$ (at most 50), whereas plots (b) and (d) show the performance of Greedy 3 when more module orderings are generated (at most 1000).

The performance of Greedy 3 on the dataset was assessed with two different stop criteria. First, when it is desirable that the computation time for Greedy 3 be of the same order of magnitude as Greedy 1 and Greedy 2 (i.e., only a fraction of a second), we set $\mu_{\max} = 50$. We will refer to this variant as Greedy 3a. When finding a higher-quality solution is important but still aiming to develop a *fast* heuristic, a time limit of one second is imposed as the second stop criterion; this variant of the algorithm is named Greedy 3b. We can see from Fig. 4 that a convenient choice for the parameter α depends on the stop criterion selected. In general, we may conclude that a good choice of the parameter α depends on the value of $\mu_{\max}$ and on the size of the instance. Below, we give specific advice for an appropriate choice of α for both variants Greedy 3a and Greedy 3b.

In Greedy 3a, only a small number of (different) module orderings are generated. Typically, the total number of possible module orderings is far higher than $\mu_{\max} = 50$, and good orderings need to be found with only a few attempts. It is to be expected that orderings that do not differ much from the ordering generated by Algorithm 1 are likely to be of reasonable quality. Fig. 4(c) shows that too much diversification ($\alpha = 1/4, 1/2$) will lead to worse solutions compared to a lower diversification level ($\alpha = 1, 2$). It is also crucial, however, to allow for a certain level of flexibility in the orderings, as illustrated by the poor performance of $\alpha = 4$ for the large instance. From Fig. 4(a) we observe that the performance of Greedy 3 is less sensitive to the level of diversification when the instance is small. This is to be expected: the total number of module orderings is also far lower. In general we infer that $\alpha = 2$ is a good choice for Greedy 3a.

Greedy 3b works with a time limit of one second; within this time we can produce close to 2000 different module orders for the large instances. Since more lists can be generated, we expect a higher diversification level to perform better; Fig. 4(d) suggests $\alpha = 1/2$ as the best choice. It is to be noted that for the small instance of 20 jobs we find only 1000 different orders for $\alpha = 1/4$ (within a time limit of one second). Higher values of α result in a high number of duplicate orderings: when $\alpha = 4$, for example, we encounter only 97 different module lists within one second.



(a) $n = 20$, $1 \leq \mu_{\max} \leq 50$

(b) $n = 20$, $1 \leq \mu_{\max} \leq 1000$

(c) $n = 120$, $1 \leq \mu_{\max} \leq 50$

(d) $n = 120$, $1 \leq \mu_{\max} \leq 1000$

Fig. 4 Relative optimality gap as a function of $\mu_{\max}$ for $\alpha \in \{1/4, 1/2, 1, 2, 4\}$ of a small instance with 20 jobs ($n20\_os6\_10$) and a large instance with 120 jobs ($n120\_os6\_5$). Plots (a) and (c) show $\mu_{\max} \leq 50$, plots (b) and (d) have $\mu_{\max} \leq 1000$.

Table 2 Average performance over 85 instances with $n \in \{10,20,30,40\}$

Algorithm	Resulting policy	CPU (s)	*ROG*
DP	Global optimum	< 0.01	0.00%
B&B	Optimal EMS	81.28	0.01%
Greedy 3b	Randomized heuristic EMS	1.00	0.13%
Greedy 3a	Randomized heuristic EMS	0.65	0.18%
Greedy 2	Deterministic heuristic EMS	< 0.01	3.21%
Greedy 1	Deterministic heuristic EMS	< 0.01	4.76%

5.2 Computational results

In Table 2, the average relative optimality gaps are given for the instances with $n \in \{10,20,30,40\}$. The branch-and-bound algorithm (B&B) cannot solve all the corresponding 120 instances to guaranteed optimality within a time limit of 30 minutes; especially larger instances and instances with a lower *OS* are more difficult to solve. We therefore report the average performance only for the 85 instances solved by the B&B. The average CPU time is less than 0.01 seconds for the dynamic-programming algorithm (DP) and for the two deterministic greedy algorithms. The average CPU time for Greedy 3a is 0.65 seconds: some time is needed for generating $\mu_{\max}$ different module orders (or for reaching a time limit of one second, whichever comes first) – this will not be the case for larger instances (see infra). For the 85 instances solved by the B&B, the average *ROG* is about 0.01%, so the expected profit is virtually the same in each case. The *ROG* of our final heuristic Greedy 3b is 0.13%. When only 50 module orderings are generated, the gap increases to 0.18%. The variation coefficient (standard deviation divided by average) of the average *ROG* of the randomized heuristics Greedy 3a and Greedy 3b in Table 2 is 0.11 and 0.015, respectively. The 3.21% gap of the best deterministic heuristic (Greedy 2) is significantly higher than the gap of Greedy 3. The average gap of Greedy 1 is yet slightly higher at 4.76%. This means that the average extra profit achieved by making a selection of jobs (not scheduling all jobs) amounts to 1.55% of the global optimum. Looking into the dataset in more depth, we observe that in 13 instances the optimal EMS policy generated by the B&B does not schedule all available jobs. In seven out of these, Greedy 2 makes the same selection, although the order is different in one of these seven instances. In three instances, Greedy 2 makes a selection that is somewhat larger than the optimal one. For three other instances, Greedy 2 fails to make a selection and schedules all available jobs, while the B&B finds an optimal policy that only schedules a subset of N. Greedy 3b finds the optimal EMS policy for all but one of these 13 instances.

For the instances with $n > 40$, the results of the heuristic can only be compared to the DP, which can solve 171 out of the 240 remaining instances without memory overrun; the results for these 171 instances are summarized in Table 3. The unsolved instances are again those with high n and low *OS*. The average runtime of the DP increases significantly when the instances become larger: it now needs 83.07 seconds on average; especially instances with a low order strength need more time. The runtime of each greedy algorithm is less than 0.01 seconds, except for Greedy 3b (which

Table 3 Average performance over 171 instances with $n \in \{50, 60, \ldots, 120\}$

Algorithm	Resulting policy	CPU (s)	*ROG*
DP	Global optimum	83.07	0.00%
Greedy 3b	Randomized heuristic EMS	1.00	0.50%
Greedy 3a	Randomized heuristic EMS	< 0.01	0.95%
Greedy 2	Deterministic heuristic EMS	< 0.01	1.67%
Greedy 1	Deterministic heuristic EMS	< 0.01	1.85%

Table 4 Average relative improvement of Greedy 3b over the remaining 69 instances that were not solved by DP, and over all 360 instances of the dataset

Algorithm	69 unsolved instances	All 360 instances
Greedy 3b	0.00%	0.00%
Greedy 3a	0.27%	0.38%
Greedy 2	0.52%	1.72%
Greedy 1	0.52%	2.18%

has an imposed time limit). Compared to Table 2, we observe that Greedy 3a is significantly faster for these larger instances: finding 50 different orderings turns out to be far easier for higher n. The gaps for Greedy 1 and Greedy 2 are slightly lower than for the instances solved by B&B (Table 2). A possible explanation might be that the optimal objective value increases with n (which is due to the way the instances were created). The improvement of Greedy 2 vis-à-vis Greedy 1 is substantially smaller, at a mere 0.18%. Since optimal EMS policies are not available for these instances, we cannot assess whether this is because our heuristic selection rule (Eq. (9)) performs worse for larger instances, or because the instances in this part of the dataset simply did not require a selection of jobs. The improvement realized by Greedy 3 is larger for small instances (Table 2) than for larger instances (Table 3), but the average relative optimality gap for our best heuristic, Greedy 3b, is still small (only a half percent).

For the instances that were not solved by the DP, we cannot determine the *ROG* because a global optimum is not known. In this case, we can only evaluate the relative gap of our algorithms with respect to the best solution found, i.e. the expected profit found by Greedy 3b. The first column of Table 4 contains the average gaps for the heuristics compared to Greedy 3b over the 69 instances that were not solved by DP. We observe an average improvement with respect to Greedy 1 and Greedy 2 of 0.52%. The selection rule seems to fail in Greedy 2 (no improvement by selection in any of these 69 instances). The improvement of Greedy 3a and Greedy 3b over Greedy 2 is realized by a reordering of the modules. Here also, no selection of jobs occurs; again we cannot evaluate whether this is because the selection rule of Eq. (9) becomes of lesser quality for higher values of n, or rather because no selection is required. The second column in Table 4 reports the averages over all 360 instances of the dataset.

Fig. 5 depicts the *ROG* of Greedy 3b as a function of n, for each of the three values of *OS*. The curves apply to the 289 instances solved by DP; each observation is the average for the solved instances of the setting considered (at most 10). No

clear patterns arise; the significant fluctuations are presumably due simply to random idiosyncrasies in the dataset.

6 Conclusions

This paper has looked into the problem of modular project scheduling on one machine (MP1), for which exact scheduling algorithms have recently been developed by Coolen et al. (2014). When only few precedence constraints are imposed, however, these algorithms either run out of memory or require increasing amounts of time when the number of activities increases. The goal of this paper was to develop a heuristic that produces 'good' schedules for such projects, i.e., schedules with a high expected profit, while requiring only very limited CPU time and computer memory.

MP1 is closely related to the series-parallel sequential testing problem. The optimal two-step procedure for the testing problem without precedence constraints is the starting point for the development of a fast heuristic procedure for MP1, in which we produce an order list that defines an elementary module-sequence scheduling policy. Four variants are proposed, in increasing order of complexity. The starting point is a simple greedy algorithm, and this is stepwise refined and complemented with a randomization step in the final variant. Based on computational experiments on a large dataset, we find that the algorithms output near-optimal solutions in negligible runtimes: the average optimality gap is 0.5% or less. We produce approximate solutions for complex instances that have not been solved by an exact algorithm within runtimes of at most one second.

A possible avenue for further research on the topic of modular project scheduling is the exploration of the multi-mode character of the modules: every module can also be seen as a 'composite' activity, which can be executed in multiple 'modes,' each mode corresponding to one possible selection of activities within the module. The lit-

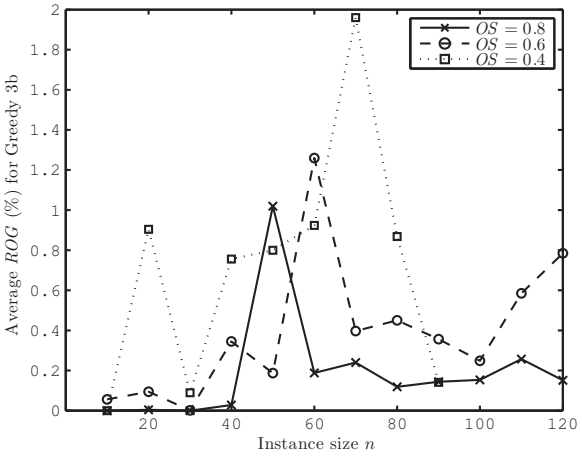


Fig. 5 Average relative optimality gap of Greedy 3b

erature on (especially project) scheduling has already looked into various multi-mode problems including time-cost trade-offs (De et al. 1995) and time-resource trade-offs (De Reyck and Demeulemeester 1998). The standard multi-mode problem is a generalization that, in addition to the time-cost and time-resource trade-offs, also covers resource-resource trade-offs and the use of multiple types of renewable and nonrenewable resources (Talbot 1982). To the best of our knowledge, however, multiple modes corresponding to multiple selections of (sub-)activities contained in the original activity (in our case: the module) have not yet been described in earlier work. From an algorithmic viewpoint, an obvious further step towards finding high-quality solutions to MP1 would be the development of a meta-heuristic procedure (for instance, tabu search or genetic algorithms, or any other meta-heuristic framework); in light of the low optimality gaps that are already achieved by the (faster) algorithms proposed in this paper, however, we have not pursued this option. For the same reason, the option of approximate dynamic programming for finding higher-quality heuristic solutions has also not been examined.

Appendix A An instance with $n = 20$

In this appendix, for illustration purposes, we describe the outputs of the algorithms proposed in this text for the instance depicted in Fig. 6. The numerical data for this instance can be found in Table 5. The instance is part of the dataset (instance name *g_n*20_*os*6_4).

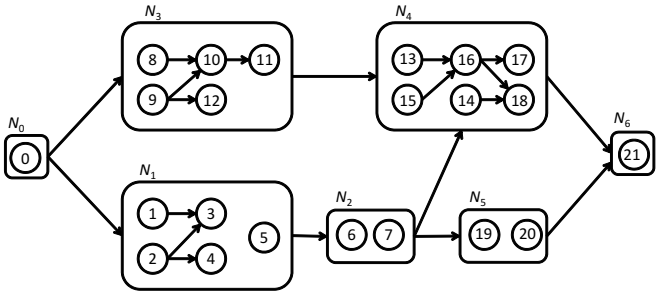


Fig. 6 Precedence network

The initial ordering generated by Greedy 1 is

$$L = (2,4,5,1,3;6,7;19,20;8,9,10,11,12;13,15,16,17,14,18),$$

which is easy to verify manually with the data from Table 5 and the pseudocode of Greedy 1. Greedy 2 removes jobs 1, 3 and 5 and the order of the modules is redetermined. In this case, the heuristic order of the modules does not change ($L' = L''$ in lines 13-16 of the pseudocode of Greedy 2), and

$$L' = (2,4;6,7;19,20;8,9,10,11,12;13,15,16,17,14,18).$$

Table 5 Costs and probabilities

k	1	2	3	4	5	6	7	8	9	10
c_k	46	10	2	12	41	32	33	15	41	16
p_k	0.961	0.891	0.895	0.836	0.912	0.977	0.844	0.833	0.922	0.978
c_k/p_k	47.9	11.2	2.2	14.4	45.0	32.8	39.1	18.0	44.5	16.4
k	11	12	13	14	15	16	17	18	19	20
c_k	15	24	17	46	22	33	45	42	14	41
p_k	0.972	0.903	0.856	0.825	0.860	0.966	0.902	0.906	0.898	0.866
c_k/p_k	15.4	26.6	19.9	55.8	25.6	34.2	49.9	46.4	15.6	47.3
V	122									

The expected profit from L and L' is approximately 14.72 and 15.05, respectively, so Greedy 2 will return L'. An optimal EMS policy found via B&B turns out to be slightly better, with an expected profit of 15.32. An optimal ordering is given by

$$(8;2,4,5,1,3;6,7;19,20;13,15,16,17,14,18);$$

this list is also found by the two implementations of Greedy 3.

References

C.Y. Baldwin and K.B. Clark. *Design Rules: The Power of Modularity*. The MIT Press, Cambridge MA, USA, 2000.

Y. Ben-Dov. A branch and bound algorithm for minimizing the expected cost of testing coherent systems. *European Journal of Operational Research*, 7(3):284–289, 1981a.

Y. Ben-Dov. Optimal testing procedures for special structures of coherent systems. *Management Science*, 27(12):1410–1420, 1981b.

Z.W. Birnbaum. On the importance of different components in a multi-component system. In P.R. Krishnaiah, editor, *Multivariate Analysis-II*, volume 1913, pages 15–26. Academic Press, New York, 1969.

R.W. Butterworth. Some reliability fault-testing models. *Operations Research*, 20 (2):335–343, 1972.

K. Coolen, W. Wei, F. Talla Nobibon, and R. Leus. Project scheduling with modular project completion on a bottleneck resource. *Journal of Scheduling*, 17(1):67–85, 2014.

S. Creemers, B. De Reyck, and R. Leus. Project planning with alternative technologies in uncertain environments. Technical Report 1314, Department of Decision Sciences and Information Management, FBE, KU Leuven, 2013.

P. De, E.J. Dunne, Gosh J.B., and Wells C.E. The discrete time/cost trade-off problem revisited. *European Journal of Operational Research*, 81:225–238, 1995.

B. De Reyck and E. Demeulemeester. Local search methods for the discrete time/resource trade-off problem in project networks. *Naval Research Logistics Quarterly*, 45:553–578, 1998.

B. De Reyck and R. Leus. R&D-project scheduling when activities may fail. *IIE Transactions*, 40(4):367–384, 2008.

A. Drexl. Scheduling of project networks by job assignment. *Management Science*, 37(12):1590–1602, 1991.

V. Jain and I.E. Grossmann. Resource-constrained scheduling of tests in new product development. *Industrial and Engineering Chemistry Research*, 38:3013–3026, 1999.

P. Jędrzejowicz. Minimizing the average cost of testing coherent systems: Complexity and approximate algorithms. *IEEE Transactions on Reliability*, R-32(1):67–70, 1983.

L.G. Mitten. An analytic solution to the least cost testing sequence problem. *The Journal of Industrial Engineering*, Januari-Februari:17, 1960.

M.L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley and Sons, 1994.

M. Ranjbar and M. Davari. An exact method for scheduling of the alternative technologies in R&D projects. *Computers and Operations Research*, 40:395–405, 2013.

T.P. Runarsson and X. Yao. Stochastic ranking for constrained evolutionary optimization. *IEEE Transactions on Evolutionary Computation*, 4(3):284–294, 2000.

C.W. Schmidt and I.E. Grossmann. Optimization models for the scheduling of testing tasks in new product development. *Industrial and Engineering Chemistry Research*, 35:3498–3510, 1996.

F.B. Talbot. Resource-constrained project scheduling problem with time-resource trade-offs: the nonpreemptive case. *Management Science*, 28:1197–1210, 1982.

T. Ünlüyurt. Sequential testing of complex systems: A review. *Discrete Applied Mathematics*, 142:189–205, 2004.